Capítulo VINTE E CINCO
Arquivos e Streams

Objetivos do Exame
Usar a API Stream com NIO.2.

Novos Métodos de Stream no NIO.2
Com a chegada dos streams ao Java, algumas operações de arquivos difíceis de implementar no NIO.2 (que frequentemente exigiam uma classe inteira) foram simplificadas.

java.nio.file.Files, uma classe com métodos estáticos que revisamos no último capítulo, adicionou operações que retornam implementações da interface Stream.

Os métodos são:

```java
static Stream<Path> find(Path start,
    int maxDepth,
    BiPredicate<Path, BasicFileAttributes> matcher,
    FileVisitOption... options)

static Stream<Path> list(Path dir)

static Stream<String> lines(Path path)
static Stream<String> lines(Path path, Charset cs)

static Stream<Path> walk(Path start,
                         FileVisitOption... options)
static Stream<Path> walk(Path start,
                         int maxDepth,
                         FileVisitOption... options)
```

Uma coisa importante a se notar é que os streams retornados são **preguiçosos (LAZY)**, o que significa que os elementos não são carregados (ou lidos) até que sejam usados. Isso é uma grande melhoria de desempenho.

Vamos descrever cada método começando com Files.list().

Files.list()

```java
static Stream<Path> list(Path dir)
            throws IOException
```

Este método itera sobre um diretório para retornar um stream cujos elementos são objetos Path que representam as entradas desse diretório.

```java
try(Stream<Path> stream =
        Files.list(Paths.get("/temp"))) {
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

Uma possível saída:

```bash
/temp/dir1
/temp/dir2
/temp/file.txt
```

O uso de **try-with-resources** é recomendado para que o método close do stream possa ser invocado e os recursos do sistema de arquivos sejam fechados.

Como você pode ver, esse método lista diretórios e arquivos no diretório especificado.
No entanto, **não é recursivo**, ou seja, **NÃO percorre subdiretórios**.

Outras duas considerações importantes sobre esse método:

- Se o argumento não representar um diretório, uma exceção será lançada.
- Este método é seguro para threads, mas é **fracamente consistente**, o que significa que, durante a iteração de um diretório, as atualizações feitas nele podem ou não ser refletidas no stream retornado.

Files.walk()

```java
static Stream<Path> walk(Path start,
                         FileVisitOption... options)
            throws IOException
static Stream<Path> walk(Path start,
                         int maxDepth,
                         FileVisitOption... options)
            throws IOException
```

Este método também itera sobre um diretório para retornar um stream cujos elementos são objetos Path que representam as entradas desse diretório.

A diferença em relação ao Files.list() é que o Files.walk() **PERCORRE** recursivamente os subdiretórios.

Ele faz isso com uma estratégia de **profundidade-primeiro (depth-first)**, que percorre a estrutura de diretórios da raiz até o fim de um subdiretório antes de explorar outro.

Por exemplo, considerando esta estrutura:

```bash
/temp/
   /dir1/
      /subdir1/
         111.txt
      /subdir2/
         121.txt
         122.txt
   /dir2/
      21.txt
      22.txt
   file.txt
```

O código a seguir:

```java
try(Stream<Path> stream =
          Files.walk(Paths.get("c:/temp"))) {
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

Gerará a seguinte saída:

```bash
/temp
/temp/dir1
/temp/dir1/subdir1
/temp/dir1/subdir1/111.txt
/temp/dir1/subdir2
/temp/dir1/subdir2/121.txt
/temp/dir1/subdir2/122.txt
/temp/dir2
/temp/dir2/21.txt
/temp/dir2/22.txt
/temp/file.txt
```

Por padrão, este método usa uma profundidade máxima de subdiretórios de Integer.MAX_VALUE.
Mas você pode usar a versão sobrecarregada que recebe a profundidade máxima como segundo parâmetro:

```java
try(Stream<Path> stream =
        Files.walk(Paths.get("/temp"), 1)) {
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

A saída será:

```bash
/temp
/temp/dir1
/temp/dir2
/temp/file.txt
```

Um valor de 0 significa que apenas o diretório inicial será visitado.

Além disso, este método **não segue links simbólicos** por padrão.

Links simbólicos podem causar um **ciclo**, uma dependência circular infinita entre diretórios.
No entanto, este método é inteligente o suficiente para detectar um ciclo e lançar uma FileSystemLoopException.

Para seguir links simbólicos, basta usar o argumento do tipo FileVisitOption (preferencialmente também utilizando o argumento de profundidade máxima), desta forma:

```java
try(Stream<Path> stream =
        Files.walk(Paths.get("/temp"),
                   2,
                   FileVisitOption.FOLLOW_LINKS)
) {
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

Assim como Files.list(), é recomendado usar Files.walk() com **try-with-resources**, e, se o argumento não representar um diretório, uma exceção será lançada.
É também considerado um método **fracamente consistente**.

Files.find()

```java
static Stream<Path> find(Path start,
        int maxDepth,
        BiPredicate<Path,BasicFileAttributes> matcher,
        FileVisitOption... options)
    throws IOException
```

Este método é semelhante ao Files.walk(), mas recebe um argumento adicional do tipo BiPredicate que é usado para filtrar os arquivos e diretórios.

Lembre-se de que um BiPredicate recebe dois argumentos e retorna um booleano. Neste caso:

- O primeiro argumento é o objeto Path que representa o arquivo ou diretório.
- O segundo argumento é um objeto BasicFileAttributes que representa os atributos do arquivo ou diretório no sistema de arquivos (como tempo de criação, se é um arquivo, diretório ou link simbólico, tamanho, etc.).
- O booleano retornado indica se o arquivo deve ser incluído no stream retornado.

O exemplo a seguir retorna um stream que inclui apenas diretórios:

```java
BiPredicate<Path, BasicFileAttributes> predicate =
    (path, attrs) -> {
        return attrs.isDirectory();
};
int maxDepth = 2;
try(Stream<Path> stream =
        Files.find(Paths.get("/temp"),
                   maxDepth, predicate)) {
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

Uma possível saída pode ser:

```bash
/temp
/temp/dir1
/temp/dir1/subdir1
/temp/dir1/subdir2
/temp/dir2
```

Assim como Files.walk(), também pode receber um FileVisitOption para visitar links simbólicos, é recomendado usá-lo com try-with-resources e lançará uma exceção se não conseguir ler um arquivo ou diretório.

Files.lines()

```java
static Stream<String> lines(Path path, Charset cs)
        throws IOException
static Stream<String> lines(Path path)
        throws IOException
```

Este método lê todas as linhas de um arquivo como um stream de Strings.

Como o stream é **preguiçoso (lazy)**, ele não carrega todas as linhas na memória, apenas a linha lida em um dado momento.
Se o arquivo não existir, uma exceção será lançada.

Os bytes do arquivo são decodificados usando o charset especificado ou com UTF-8 por padrão.

Por exemplo:

```java
try(Stream<String> stream =
            Files.lines(Paths.get("/temp/file.txt"))) {
        stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

No Java 8, um método lines() também foi adicionado a java.io.BufferedReader:

```java
Stream<String> lines()
```

O stream é preguiçoso, e seus elementos são as linhas lidas do BufferedReader.

Pontos-chave

- No Java 8, novos métodos que retornam implementações da interface Stream foram adicionados à classe java.nio.file.Files.
- Os streams retornados são **preguiçosos (LAZY)**, o que significa que os elementos não são carregados (ou lidos) até que sejam usados.
- É recomendável o uso de **try-with-resources** com esses métodos para que o método close do stream possa ser invocado e os recursos do sistema de arquivos sejam fechados.
- Files.list() itera sobre um diretório para retornar um stream cujos elementos são objetos Path que representam as entradas desse diretório.

Este método lista diretórios e arquivos do diretório especificado. No entanto, **não é recursivo**, ou seja, **NÃO percorre subdiretórios**.

- Files.walk() também itera sobre um diretório com uma estratégia de profundidade-primeiro para retornar um stream cujos elementos são objetos Path que representam as entradas desse diretório.
- A diferença em relação ao Files.list() é que Files.walk() **PERCORRE recursivamente os subdiretórios**. Você também pode passar a profundidade máxima de travessia e uma opção para seguir links simbólicos.
- Files.find() é semelhante ao Files.walk(), mas recebe um argumento adicional do tipo BiPredicate<Path,BasicFileAttributes> que é usado para filtrar os arquivos e diretórios.
- Files.lines() lê todas as linhas de um arquivo como um stream de Strings **sem carregá-las todas na memória**.

Autoavaliação

1. Dada a seguinte estrutura e classe:

bash Copiar Editar

```
/temp/
   /dir1/
      1.txt
   0.txt
```

java Copiar Editar

```java
public class Question_25_1 {
    public static void main(String[] args) {
        try(Stream<Path> stream =
                Files.walk(Paths.get("/temp"), 0)) {
            stream.forEach(System.out::println);
        } catch(IOException e) { }
    }
}
```

Qual é o resultado?

A. /temp
B. /temp/dir1
/temp/0.txt
C. /temp/0.txt
D. Nada é impresso

---

2. Qual das afirmações a seguir é verdadeira?

A. Files.find() tem uma profundidade padrão de subdiretórios igual a Integer.MAX_VALUE.
B. Files.find() segue links simbólicos por padrão.
C. Files.walk() segue links simbólicos por padrão.
D. Files.walk() percorre subdiretórios recursivamente.

---

3. Qual das seguintes opções é equivalente a:

```java
Files.walk(Paths.get("."))
    .filter(p -> p.toString().endsWith("txt"));
```

A.

```java
Files.list(Paths.get("."))
    .filter(p -> p.toString().endsWith("txt"));
```

B.

```java
Files.find(Paths.get("."),
    (p,a) -> p.toString().endsWith("txt"));
```

C.

```java
Files.find(Paths.get("."), Integer.MAX_VALUE,
     p -> p.toString().endsWith("txt"));
```

D.

```java
Files.find(Paths.get("."), Integer.MAX_VALUE,
    (p,a) -> p.toString().endsWith("txt"));
```

---

4. Qual é o comportamento de Files.lines(Path) se o objeto Path representar um arquivo que não existe?

A. Ele retorna um stream vazio.
B. Ele cria o arquivo.
C. Ele lança uma IOException quando o método é chamado.
D. Ele lança uma IOException quando o stream é usado pela primeira vez.